Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2024 SCPE. Volume 25, Issues 6, pp. 4886–4897, DOI 10.12694/scpe.v25i6.3393

PRE-DNNOFF: ON-DEMAND DNN MODEL OFFLOADING METHOD FOR MOBILE EDGE COMPUTING

LIN ZUO*

Abstract. Deep Neural Networks (DNNs) are critical for modern intelligent processing but cause significant latency and energy consumption issues on mobile devices due to their high computational demands. Moreover, different tasks have different accuracy demands for DNN inference. To balance latency and accuracy across various tasks, we introduce PreDNNOff, a method that offloads DNNs at a layer granularity within the Mobile Edge Computing (MEC) environment. PreDNNOff utilizes a binary stochastic programming model and Genetic Algorithms (GAs) to optimize the expected latency for multiple exit points based on the distribution of task inference accuracy and layer latency regression models. Compared to the existing method Edgent, PreDNNOff has achieved a reduction of about 10% in the expected total latency, and due to the consideration of different tasks' varying requirements for accuracy, it has a broader applicability.

Key words: Computation offloading, deep neural networks, intelligent Internet of Things (IoT), mobile edge computing (MEC)

1. Introduction.

With the rapid advancement of deep neural networks (DNNs), which serve as a cornerstone technology supporting modern intelligent processing [1], they have become the most commonly employed machine learning technique and are increasingly gaining popularity. However, due to the substantial computational requirements typically associated with DNN-based applications, they cannot be well-supported by today's mobile devices in terms of reasonable latency and energy consumption. Therefore, they are usually trained and executed in cloud environments. In other words, input data generated by mobile devices is transmitted to the cloud for processing, and the results are returned to the mobile devices after inference. Nevertheless, for this cloud-centric approach to data processing, if the volume of input data becomes excessively large, the network communication between mobile devices and the cloud can lead to intolerable execution delays, significantly impacting the user experience. To mitigate the latency of cloud-centric approaches, a superior solution is to introduce Mobile Edge Computing (MEC) [2]. Chen et al. [6] elucidated how Mobile Edge Computing (MEC) [3] overcomes the inherent limitations of Mobile Cloud Computing (MCC) , particularly the issue of prolonged latency between mobile devices and remote clouds. They pointed out that the high latency and energy consumption resulting from the transmission of a substantial amount of data generated by DNN models over wireless networks to the cloud made the existing work in the MEC environment unsuitable for DNN-based applications. Consequently, they proposed DNNOff, a novel method for DNN offloading in MEC environments. DNNOff translates DNN-based applications into target structures that are easier to offload while using a random forest regression model to predict the latency of offloading schemes. Based on the predictive model, DNNOff can determine which portions should be transferred to MEC servers.

However, DNNOff only considers optimizing for latency and ignore the problem of inference accuracy. Some applications require both low latency and inference accuracy. For instance, in the field of public safety, real-time facial recognition in video surveillance requires the ability to display results in real-time for law enforcement personnel to quickly locate and track individuals while ensuring recognition accuracy to avoid hindering their work. Hence, there is a need for a balance between latency and accuracy. Combining the BranchyNet structure proposed by Teerapittayanon et al. [5] of early exit mechanism [4] with DNNOff can reduce latency while maintaining a certain level of accuracy.However, in real-world scenarios, different types of tasks have varying accuracy requirements, and the types of these tasks are usually random.

*The University of Science and Technology of China, School of Information Scicnce and Technology, Hefei 230000, China (zlys13579zl@mail.ustc.edu.cn)

Based on the consideration above, we propose a preemptive optimization method named Pre-DNNOff, built upon the early exit mechanism offloading strategy. We model this method as a binary stochastic programming model based on the multi-task inference accuracy distribution with multiple exit points' expected latency. Below is a detailed explanation of the model. In the training and deployment phase, Pre-DNNOff generates regression prediction models for different DNN layers on mobile devices and edge servers considering input, output, and execution time. For instance, the execution time of fully connected layers can be linearly expressed by the size of input and output data. Then, in the modeling phase, Pre-DNNOff combines the historical requirements for inference accuracy of tasks, channel bandwidth in the MEC environment, and the prediction models of DNN layers to calculate the expectation of total latency. In the encoding and solving phase, binary variables represent whether a layer is offloaded to the edge, thus obtaining the encoding for each offloading scheme. Finally, genetic algorithms are used for searching space encoding and problem solving. Through this method, we achieve the minimum expected delay while meeting the accuracy requirements of different tasks. However, as the number of branches in BranchyNet increases and the number of layers within the branches grows, the time complexity of solving this problem using genetic algorithms may rise exponentially. This makes it impossible to obtain the optimal offloading scheme within polynomial time. Additionally, in more complex MEC environments, the distribution of task accuracy requirements and the regression model for accuracy layer latency also incur additional costs to obtain.

The main contributions of this paper are summarized as follows:

1. We proposed a prediction-based optimization method called Pre-DNNOff and utilized Genetic Algorithm to solve it. Compared to existing methods, PreDNNOff achieves lower expected latency and broader applicability.
2. It is represented as a binary stochastic programming model that considers the distribution of multi-task inference accuracy and joint expected latency with multiple exit points. Furthermore, we optimize the BranchyNet structure by using a regression model for DNN layer latency. This results in a reduction in model complexity.

2. Related Works. Due to limitations in storage space, battery life, and computational capability [7], mobile devices generally cannot directly execute computational tasks. To address this issue, computation offloading based on Mobile Edge Computing (MEC) [8] has become the most widely used technique. Chen et al. [6] elucidated how MEC [9] overcomes the inherent limitations of Mobile Cloud Computing (MCC) [10], particularly the issue of prolonged latency between mobile devices and remote clouds. They pointed out that the high latency and energy consumption resulting from the transmission of a substantial amount of data generated by DNN models over wireless networks to the cloud made the existing work in the MEC environment unsuitable for DNN-based applications. Consequently, they proposed DNNOff, a novel method for DNN offloading in MEC environments. DNNOff translates DNN-based applications into target structures that are easier to offload while using a random forest regression model to predict the latency of offloading schemes. Based on the predictive model, DNNOff can determine which portions should be transferred to MEC servers. Lin et al.[11], building upon DNNOff's DNN structure, proposed a self-adaptive particle swarm optimization (PSO) algorithm that utilizes genetic algorithm (GA) operators (PSO-GA) to reduce system costs resulting from data transmission and layer execution, all while adhering to the deadline constraints of all DNN-based applications. However, the aforementioned work overlooked the issue of DNN inference accuracy.

Teerapittayanon et al. [5] introduced a novel deep network architecture called BranchyNet. It enhances the existing network architecture by adding additional branch classifiers. Through these branch classifiers, this architecture allows a significant portion of test samples' prediction results to exit the network early through these branches, by which time the samples can be confidently inferred. This early exit mechanism reduces latency while ensuring a certain level of accuracy. Li et al. [12] proposed Edgent, which is based on BranchyNet and is a collaborative and on-demand DNN collaborative inference framework with device-edge synergy. Edgent accomplishes two functions. First, it adaptively divides DNN computation between devices and the edge, enabling real-time DNN inference using nearby hybrid computing resources. Second, it allows for early exits at appropriate intermediate DNN layers to meet DNN latency deadlines while maximizing DNN inference accuracy.

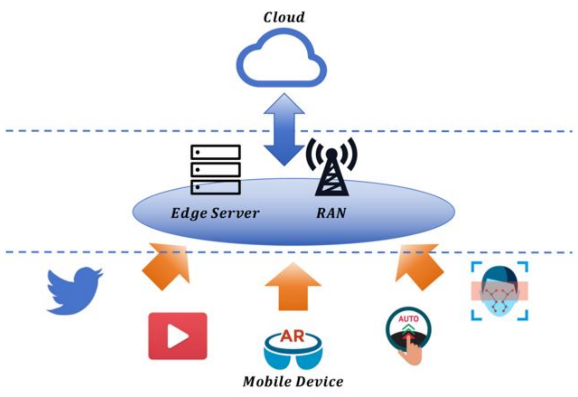

Fig. 3.1: Mobile Edge Computing Architecture: Storage and computational servers deployed at the RAN that enables a range of services to the network users.

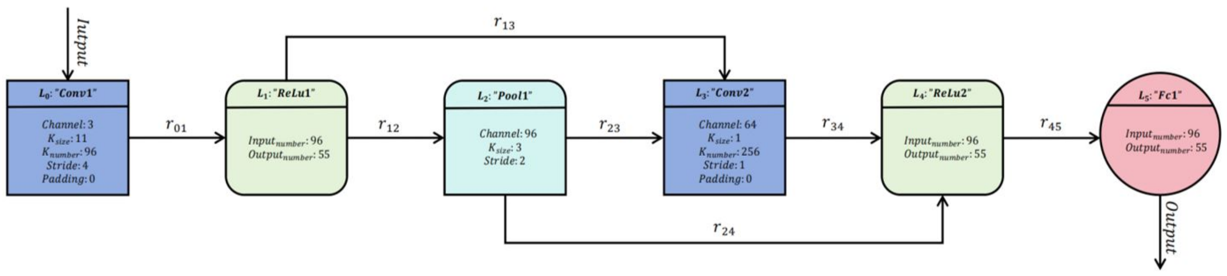

Fig. 3.2: Example of the DNN graph processed by DNNOff.

3. Preliminary.

3.1. Information about MEC.

As a representative of emerging IT technologies, MEC is a product of the convergence of Information and Communications Technology (ICT). It combines technologies such as Software Defined Networking/Network Function Virtualization (SDN/NFV), big data, artificial intelligence, and more. The 5G network is becoming a critical infrastructure for digital transformation in various industries, and MEC plays a pivotal role in supporting the development of services such as high-definition video, VR/AR, industrial Internet, and connected vehicles [2]. Figure 3.1 illustrates the architecture of mobile edge computing, highlighting its role in enabling various services for network users. MEC also brings several benefits to end-users [13]. Users can offload their compute-intensive tasks to edge servers. By offloading computation and accessing locally cached content, end-users can significantly reduce end-to-end latency. Since mobile users are battery-powered devices, they can also leverage MEC to conserve energy consumption. When content is cached locally in the RAN (i.e., available at lower propagation distances or even in a single-hop), video data packets can be delivered with minimal latency and relatively fewer packet delay variations, thereby improving connectivity and enhancing stability. With the power of edge computing, mobile users can run new applications, including compute-intensive artificial intelligence applications.

3.2. Structure of Common DNNs.

A typical DNN structure consists of a series of interconnected layers[14], with each layer containing a certain number of nodes. Each node represents a neuron that performs operations on its inputs and produces an output. The input layer of nodes is set by the raw data, while the output layer determines the category of data.

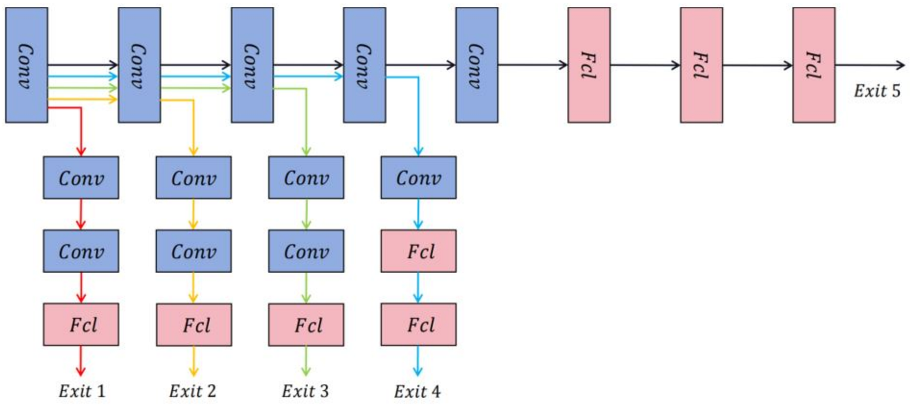

Fig. 3.3: An AlexNet with five exit points.

Figure 3.2 displays an example of the DNN graph processed by DNNOff. In the graph, different types of layers are represented by distinct shapes of different colors. Specifically, deep blue represents convolutional layers, which translate images into feature maps with learned filters. Green represents activation layers, which are non-linear functions. These functions take a feature map as input and produce an output with the same dimensions. Light blue represents pooling layers, which are typically divided into general pooling, average pooling, or maximum pooling. Red represents a fully connected layer, which calculates the weighted sum of inputs by learning the weights. The top of the square contains the layer's number and name, and the bottom of the square contains the layer's parameters. For example, " L_0" corresponds to "Layer 0," "Conv1" indicates that this layer is of type convolution, and "Channel: 3" signifies that the parameter "Channel" has a value of 3. Black arrows represent data flows, such as "r_{01}", which signifies the output from Layer 0 to Layer 1.

3.3. BranchyNet Architecture. BranchyNet is a novel deep network architecture introduced by Teerapittayanon et al. [5]. A BranchyNet network consists of an entry point and one or more exit points. A branch is a subset of the network that comprises consecutive layers that do not overlap with other branches, followed by an exit point. This network can be considered as being composed of a main branch (the original network) and side branches (additional networks). Figure 3.3 shows an example of an AlexNet [15] with five exit points. For simplicity, only the convolutional layers and fully connected layers of this network are shown. In the figure, starting from the lowest branch and moving to the highest branch, each branch along with its associated exit points is sequentially numbered, starting from 1. The input data first enters the network's input layer, where some preprocessing steps, such as normalization, may be included. Then the data passes through a series of convolutional and pooling layers for feature extraction. These layers are responsible for extracting useful features from the input data, such as edges, textures, and shapes. At multiple points in the network, the model assesses whether the features currently extracted are sufficient to make an accurate classification decision. If certain conditions are met, the model can choose not to delve deeper into the network and instead classify at the current layer. If the data passes the early exit point, it will enter one or more fully connected layers and ultimately reach the classification layer. The classification layer usually consists of one or more softmax layers, which are used to output the probability of each category. With the increasing number of the network layers,the classification accuracy will improve simultaneously.

4. System Model.

4.1. Model Overview. Figure 4.1 is the overview of Pre-DNNOff.There are two phases in Pre-DNNOff, including training & deployment phase and modeling & solving phase.During the training and deployment phase, Pre-DNNOff initializes two modules:

1. It performs an analysis of mobile devices, edge servers, and cloud servers, generating performance

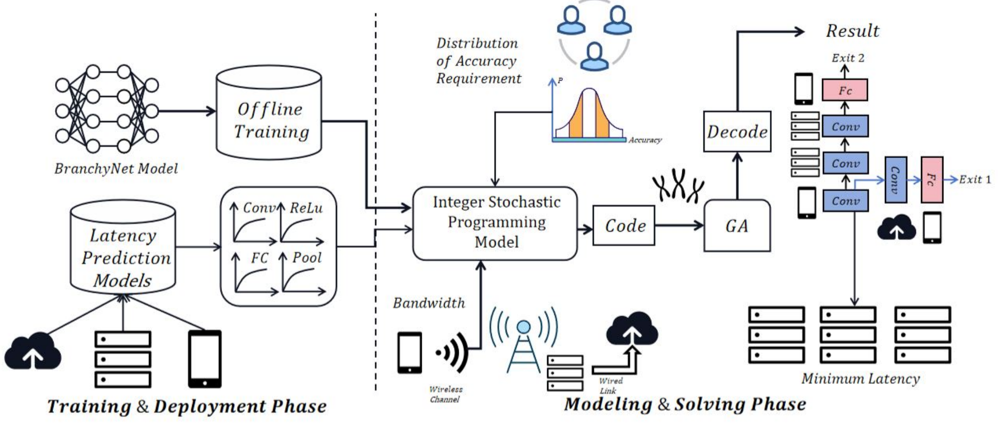

Fig. 4.1: The overview of Pre-DNNOff.

prediction models based on regression for different types of DNN layers, such as convolution and pooling (as discussed in 4.2).

2. It trains DNN models with BranchyNet architecture to implement the early exit mechanism. As mentioned in 3.2, DNNs have a large number of parameters, and typically, offline training is conducted in powerful cloud servers. Additionally, since performance prediction models for different types of DNN layers depend on the infrastructure, offline fitting of prediction models is also necessary.

During the modeling and solving phase, Pre-DNNOff constructs a binary stochastic programming model based on the distribution of multi-task inference accuracy and joint expected latency with multiple exit points. It encodes this model and employs Genetic Algorithms (GA) for solving.

1. Historical information on task (or user) requirements for inference accuracy and surveys are used to fit distributions, along with channel bandwidth in the MEC hybrid environment. Combined with the previously trained BranchyNet network and prediction models for different types of DNN layers, a binary stochastic programming model based on the distribution of multi-task inference accuracy and joint expected latency with multiple exit points is constructed.
2. Results from the entire search space are encoded as "chromosomes" using a single mapping relationship, and GA is employed to solve the problem. Finally, we can get the shortest expectation of the total latency.

4.2. Layer Latency Prediction. While complex DNNs may consist of a vast number of layers, the categories of layers that compose a DNN are extremely limited. Typically, DNN layer categories include convolution, ReLU, pooling, normalization, dropout, fully connected, and linear layers. Calculating the total execution time of a DNN involves computing the execution time for each layer individually and then summing them all.

Li et al. [12] conducted experiments to explore various variables (e.g., input data size, output data size) that determine the latency of different layers. These variables are listed in Table 4.1. We utilized a Raspberry Pi microcomputer and a desktop computer for this purpose. We established a regression model with the above variables as independent variables and layer execution time as the dependent variable. This allowed us to predict the execution time for each layer based on its characteristics. Additionally, we took into account the initial loading time of the DNN model onto the mobile device and edge server, as well as the parsing time when the final data is input to the mobile device. Furthermore, we included the size of the DNN model as an input parameter to predict model loading and parsing times. The regression models for each type of layer are presented in Table 4.2 (sizes are in bytes, and latency is in milliseconds).

Table 4.1: The variables of regression models

LayerType	Variable
Convolution	amount of input feature maps(x_1) , $(\frac{K_{size}}{Stride})^2 \cdot K_{number} \cdot Chnannel(x_2)$
ReLu	input data size(x)
Pooling	input data size (x_1), output data size (x_2)
Local ResponseNormalization	input data size (x)
Dropout	input data size (x)
Fully-Connected	Input data size (x_1),output data size (x_2)
Model Loading	model size (x)
Model Parsing	model size (x)

Table 4.2: Regression models of each layer type

Layer	Edge Server model	Mobile Device model
Convolution	$t = 6.03\times10^{-5}x_1+1.24\times10^{-4}x_2+1.89\times10^{-1}$	$t = 6.13\times10^{-3}x_1+2.67\times10^{-2}x_2-9.909$
ReLu	$t = 5.6 \times 10^{-6}x + 5.69 \times 10^{-2}$	$t = 1.5 \times 10^{-5}x + 4.88 \times 10^{-1}$
Pooling	$t = 1.63\times10^{-5}x_1+4.07\times10^{-6}x_2+2.11\times10^{-1}$	$t = 1.33\times10^{-4}x_1+3.31\times10^{-5}x_2+1.657$
Local ResponseNormalization	$t = 6.59 \times 10^{-5}x + 7.80 \times 10^{-2}$	$t = 5.19 \times 10^{-4}x + 5.89 \times 10^{-1}$
Dropout	$t = 5.23 \times 10^{-6}x + 4.64 \times 10^{-3}$	$t = 6.59 \times 10^{-5}x + 5.25 \times 10^{-2}$
Fully-Connected	$t = 1.07\times10^{-4}x_1-1.83\times10^{-4}x_2+1.64\times10^{-1}$	$t = 9.18\times10^{-4}x_1+3.99\times10^{-3}x_2+1.169$
Model Loading	$t = 1.33 \times 10^{-6}x + 2.182$	$t = 4.49 \times 10^{-6}x + 82.136$
Model Parsing	\	$t = 3.48 \times 10^{-6}x + 4.253$

4.3. Integer Stochastic Programming Model. First, we convert the already offline-trained BranchyNet network into a graph $G =$ $(L,\ R)$ that contains layer information and the BranchyNet topology structure, where $L\!=\{L_1, L_2, ..., L_P\}$ represents a set of DNN layers corresponding to P exit points of BranchyNet. We number the exit points of BranchyNet according to the method in Part Structure of Common DNNs.Each $L_k\!=\left\{l_k^0, l_k^1, ..., l_k{}_k^N\right\}$ represents a set of N_k layers in the k-th exit point of BranchyNet. The i-th layer in the k-th exit point is represented as:

$$l_k^i = \left\langle Ltype_k^i, Var_k^i \right\rangle \tag{4.1}$$

Here, $Ltype_k^i\!= \{0, 1, 2, 3, 4, 5\}$ corresponds to the type of the i-th layer in the k-th exit point, where 0 represents Convolution, 1 represents ReLu, 2 represents Pooling, 3 represents Local Response Normalization, 4 represents Dropout, and 5 represents Fully-Connected. Var_k^i represents the variables in the regression models for different layer types as established in Part Layer Latency Prediction.R represents the set of data flows between layers, corresponding to the set of edges in the graph. $r_k^{ij}\!\in\!R$ represents the data flow from l_i to l_j in the k-th exit point, where $\forall i,\ j =\ 0,\ 1,\ ...,\ N$, and i $\neq$ j. Here, we consider BranchyNet as having both a main branch and side branches with a simple chain structure, so we can further simplify r_k^{ij}. Specifically, we define $Output_k^i$ as the output of the i-th layer in the k-th exit point. When $0\!\leq\! i\!<\!N_k, Output_k^i$ is the input to the i+1-th layer in the k-th exit point; i = N_k indicates the final output of the k-th exit point.

In a typical MEC heterogeneous network environment, there are multiple mobile terminals, a base station (BS) equipped with multiple edge servers (ES), and a cloud server (CS) wired connected to the BS. We consider a simplified MEC mixed environment, assuming that all mobile terminals have similar wireless channel performance, and there is only one edge server. Tasks generated by mobile terminals can be processed locally on the mobile device, offloaded to ES via wireless channels allocated by the BS, or further offloaded from BS to CS via a wired link to the CS. ES and CS allocate their computational resources to offloaded tasks. The bandwidth of wireless channels is denoted as W, and the signal-to-noise ratio is denoted as SNR. According to the Shannon-Hartley formula, the transmission rate of wireless channels is given by:

$$v_{wireless} = W\log_2(1 + SNR) \tag{4.2}$$

For the wired transmission part, we can similarly define $v_{wired\ link}$.

In Part Layer Latency Prediction, we established regression models for each layer type, and therefore, we obtained layer latency times represented as:

$$T_{Stype}\left(l_k^i\right) = f\left(Stype, Var_k^i\right) \tag{4.3}$$

Here, Stype = {0, 1, 2} represents the type of selected server: 0 for mobile, 1 for edge, and 2 for cloud. We also define a binary variable $Sel_{Stype}^{l_k^i}$ to represent whether the i-th layer in the k-th exit point is offloaded to Stype. We assume a serial processing model in which a server can only execute one layer at a time, and entire layer processes on a single server. Thus, $\sum_{Stype=0,1,2} Sel_{Stype}^{l_k^i} = 1$.

The total execution time for the k-th exit point can be represented as:

$$T_{execution}^k = \sum_{i=0}^{N_k-1} \sum_{Stype=0,1,2} Sel_{Stype}^{l_k^i} T_{Stype}\left(l_k^i\right) \tag{4.4}$$

Data transmission from the local end to the cloud requires passing through the edge, and then transmitted to the cloud. Therefore, the transmission time is:

$$T_{transmission}^k = \sum_{i=1}^{N_k} \left(\frac{\left| Sel_0^{l_k^i} - Sel_0^{l_k^{i-1}} \right|}{v_{wireless}} + \frac{\left| Sel_2^{l_k^i} - Sel_2^{l_k^{i-1}} \right|}{v_{wiredlink}} \right) \cdot Output_k^i \tag{4.5}$$

So, the total latency for the k-th exit point is represented as:

$$Latency_k = T_{execution}^k + T_{transmission}^k \tag{4.6}$$

Since the output data needs to be transmitted to mobile devices for storage and parsing, we have $Sel_0^{l_k^{N_k}} = 1$.

To predict the future allocation of computational resources in the MEC mixed environment, we need to investigate the requirements of tasks (or users) for inference accuracy. Let $d(ac)$ be the probability density function of the distribution $D(ac)$ that represents the inference accuracy requirements as a random variable ac. We can obtain the inference accuracy Acc(k) for each k-th exit point from the trained BranchyNet network on the test set. Our goal is to minimize the inference latency under the inference accuracy requirements. Therefore, this optimization problem can be formulated as:

$$\min_{Sel_{Stype=0,1,2,k\in\{1,2,\ldots,P\}}^{l_i^k}, ac} \quad Latency \tag{4.7}$$

Subject to:

$$C_1 : Sel_{Stype=0,1,2}^{l_i^k} = \{0,1\},$$

$$C_2 : Acc(k) \geq ac,$$

$$C_3 : \sum_{Stype=0,1,2} Sel_{Stype}^{l_k^i} = 1,$$

$$\forall k \in \{1, 2, \ldots, P\}.$$

The above optimization problem is a stochastic programming problem. Optimizing the objective function's expected value under constraints is known as an expectation optimization problem. In this case, we consider using the expectation optimization model to solve this stochastic programming problem, aiming to minimize the expected inference latency under given accuracy requirements. For a given accuracy requirement, we need to ensure that the accuracy achieved at the k-th exit point slightly exceeds the requirement. In other words, $Acc(k-1) < ac \leq Acc(k)$ (for $k \neq 1$, P). We define $Acc(0) = 0$ and Acc(P+1) = 1. Therefore, the expected latency is given by:

$$\sum_{k=1}^{P+1} Latency\,(k) \cdot \int_{Acc(k-1)}^{Acc(k)} d\,(x)\,dx \tag{4.8}$$

Hence, this stochastic programming problem can be transformed into a binary nonlinear programming problem:

$$\min_{Sel^{l_i^k}_{Stype=0,1,2,k\in\{1,2,\ldots,P\}}} \sum_{k=1}^{P+1} Latency\,(k) \cdot \int_{Acc(k-1)}^{Acc(k)} d\,(x)\,dx \tag{4.9}$$

$$s.t. : C_1 : Sel^{l_i^k}_{Stype=0,1,2} = \{0, 1\},$$

$$C_2 : \sum_{Stype=0,1,2} Sel^{l_k^i}_{Stype} = 1,$$

$$\forall k \in \{1, 2, \ldots, P\}.$$

4.4. Genetic Algorithm.

1. *Problem Encoding*: Encoding is the mapping from a solution to a genotype, i.e., the method to transform feasible solutions from the problem space to the search space of the genetic algorithm. Encoding strategies typically need to satisfy three principles: (Completeness) Every candidate solution can be encoded into a chromosome in the population. (Nonredundancy) Each candidate solution corresponds to only one chromosome in the population. (Viability) Each encoded chromosome represents a feasible solution in the problem space. In this paper, we consider encoding the offload binary variables $Sel^{l_k^i}_{Stype}$ into chromosomes by concatenating them according to the branching layers of BranchyNet, forming a 01 sequence that represents an offloading scheme, satisfying the first principle. We assume a serial processing model and that BranchyNet has both a main branch and side branches with a chain structure, so there are no issues with overlapping processing times for different layers on the same server or conflicts in the offloading of the same layer to different servers. Therefore, the mapping between chromosomes and candidate solutions is injective, satisfying the second and third principles. Figure 4.2 illustrates the specific way of transforming offloading variables into chromosomes.
2. *Fitness Function*: The fitness function indicates the quality of an individual or solution. Different problems require different definitions for the fitness function. In our case, we use the expected latency as the fitness function. So, the smaller the value of the fitness function, the better the corresponding chromosome's offloading scheme. As the algorithm iterates, competition between individuals in the population gradually weakens as their fitness becomes closer, potentially causing the population to converge to a local optimum. To address this issue, we need to perform fitness scaling. Here, we use linear fitness scaling.
3. *Update Strategy*: The update strategy in genetic algorithms involves the use of genetic operators, which include selection, crossover (recombination), and mutation.

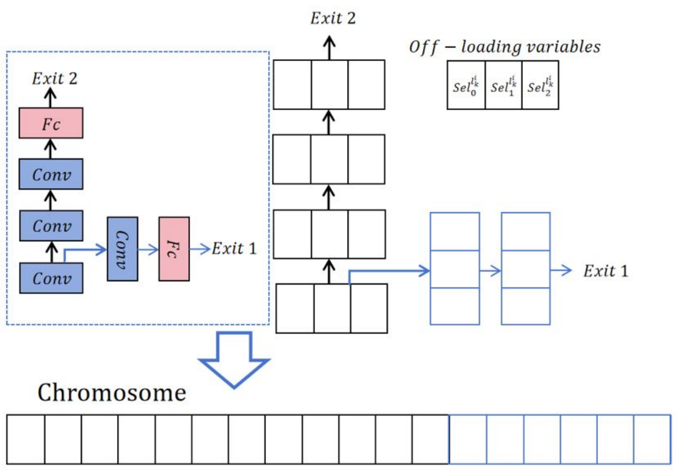

Fig. 4.2: A concrete way to transform unloaded variables into chromosomes

- *Selection*: The selection operation chooses a set of individuals from the old population with a certain probability to form a new population for the next generation. The probability of selecting an individual is related to its fitness value; the higher the fitness value, the greater the chance of selection. Here, we use roulette wheel selection. If there are M individuals in the population, and the fitness of individual i is denoted as f_i, then the probability of selecting individual i is given by

$$P_i = \frac{f_i}{\sum_{k=1}^{M} f_k} \tag{4.10}$$

  Once the selection probabilities are determined, random numbers between 0 and 1 are generated to decide which individuals participate in mating. Individuals with higher selection probabilities have a greater chance of being selected, potentially leading to their genes being passed on to more offspring.
- *Crossover (Recombination)*: The crossover operation involves selecting two individuals randomly from the population and combining their chromosomes to create new offspring with a mix of their parents' features. In our case, we use a single-point crossover operator. In this operator, a random crossover point is chosen, and genes are exchanged between the two parent chromosomes at that point. It's worth noting that the smallest unit we crossover is a group of layer offloading variables, specifically the variables where Stype=0,1,2 for the same layer.
- *Mutation*: The mutation operation is performed to prevent the genetic algorithm from getting stuck in local optima during the optimization process. In our case, we perform single-point mutation at the level of layer offloading variable groups. Specifically, we randomly select one variable with a value of 0 in the layer offloading variable group and change it to 1, while setting all other variables to 0.

  These genetic operators collectively drive the evolution of the population over generations, with selection favoring individuals with better fitness, crossover mixing their features, and mutation introducing genetic diversity.

5. Experiment Results and Analysis.

5.1. Experimental Setup. To simplify the experiments, we only consider mobile and edge computing scenarios. The analyses mentioned earlier are also applicable in this case, with the adjustment of the number of variables in the layer offloading variable group and the removal of latency variables offloaded to the cloud.

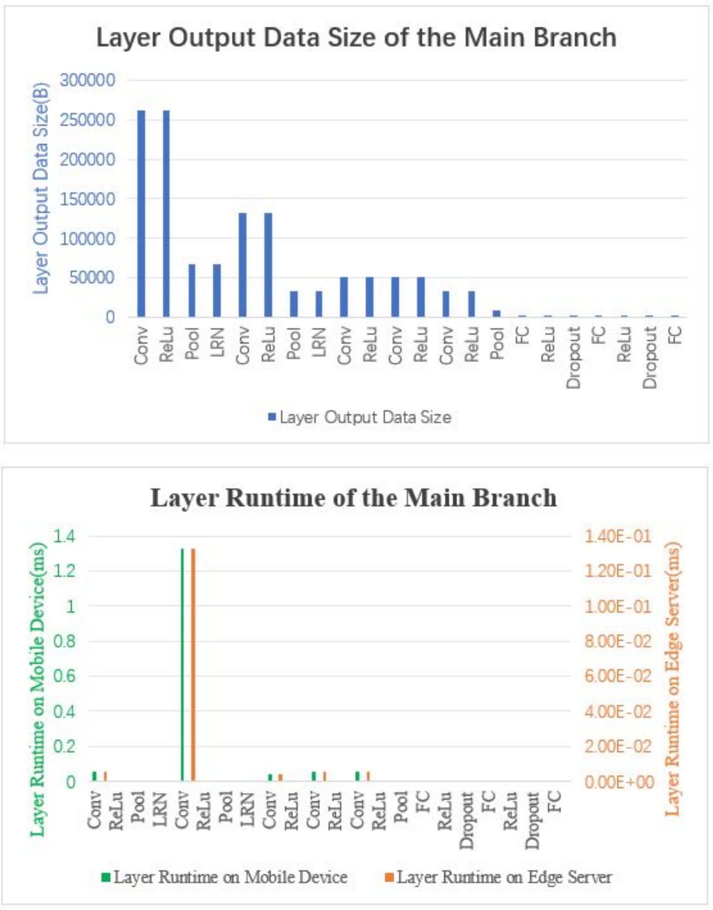

Fig. 5.1: Output data size of the main branch of the DNN and the running time on a Raspberry Pi3B micro-computer and a laptop respectively

Similar to the work of Li et al. [12], we use a laptop to simulate the edge server, equipped with an 8-core 3.6GHz processor and 8GB of memory. For mobile devices, we use a Raspberry Pi 3B mini-computer, which features a 1.2GHz 64-bit quad-core ARM processor and 1GB of RAM.

For the channel between the mobile and edge, we consider the most common LTE (Long-Term Evolution) standard. While the FDD-LTE (Frequency Division Duplex LTE) standard has a bandwidth of 2×20MHz, theoretically supporting downlink rates of 150Mbps and uplink rates of 40Mbps, real-world user experiences often yield lower rates, approximately around 93Mbps, as reported in official surveys . Taking into account interference from buildings and other signals, we assume a rate of 74Mbps.

Regarding the BranchyNet model, we follow the work of Teerapittayanon et al. [5]. We train and test a modified version of the standard image recognition base model, AlexNet, on the CIFAR-10 dataset . Additionally, we choose Chainer as the deep learning framework due to its excellent support for branch DNN structures.

5.2. Experimental Results and Analysis. We have demonstrated a DNN model with a BranchyNet structure, modified from the standard AlexNet, in Figure 3.3 (only convolution layer and fully connected layer are drawn). Figure 5.1 shows the output parameters of the main branch of this Branchy AlexNet and the running times on Raspberry Pi 3B microcomputer and laptop, with side branches not drawn. The four side branches respectively have 7,7,8,8 layers. In the test set, the accuracy of inference results from different exits

Table 5.1: Accuracy of each exit point

Exit Point	1	2	3	4	5
Accuracy(%)	0.524	0.582	0.687	0.749	0.784

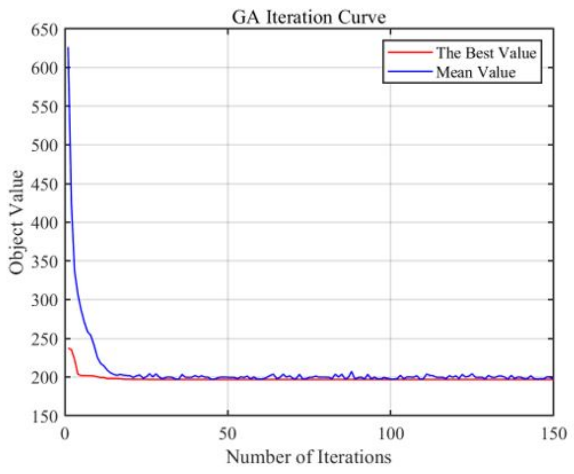

Fig. 5.2: GA Iteration Curve in One Experiment

is as shown in the table. AlexNet [15] achieved error rates of 37.5% and 17.0% on top-1 and top-5 test sets on ILSVRC-2010, respectively. We take the top-1 accuracy of 62.5% as the mean expected inference accuracy, assuming a normal distribution. The accuracy of our DNN model's output results ranges between 52.4% and 78.4%, meaning $P\ (52.4\% < acc \leq 78.4\%) \geq 99.7\%$. So, we take the standard deviation (σ) as 0.053.

In the experiment, we set the iteration times of the GA to 50. Since GA belongs to heuristic algorithms, although the basic configuration of each experiment is consistent, the offloading results between different experiments may vary. Hence, we need to conduct multiple experiments to avoid the results falling into local optimums. Figure 5.2 shows a GA iteration curve in one experiment. We can see that the optimal solution is obtained around the 12th iteration, and the mean value of the population becomes stable at the 21st iteration. The scale of our problem is not large. After conducting 20 experiments, the offloading scheme of DNN and the minimum expected total latency were all the same. Hence, we can be fairly certain that the optimal solution of the problem is the result consistently obtained in these 20 experiments.

By using Pre-DNNOff to process the aforementioned DNN model,we can get the shortest total expected latency is 188.93ms. Under the same conditions, the total expected latency of the offloading scheme obtained using Edgent is 207.11ms. PreDNNOff has achieved approximately a 10% reduction in latency compared to Edgent. Moreover, while Edgent is only applicable in situations where the task's accuracy requirements are known, PreDNNOff can address scenarios where the task's accuracy requirements are uncertain.

6. Conclusion. In this paper, we introduce a new DNN offloading model, Pre-DNNOff, which seeks to strike a balance between inference latency and accuracy. Upon solving the stochastic programming problem transformed by Pre-DNNOff using GAs, we derive a lower latency for mobile edge environments with uncertain task requirements of latency. However, in this study, we only consider applications in a simplified IoT setting. In the future, we plan to apply Pre-DNNOff in more complex environments. We aim to optimize Pre-DNNOff further by treating the parameters of the transmission bandwidth and accuracy requirement distributions as random variables, enabling more realistic adjustments of computational resources. Besides,we will design an algorithm that can still be completed within polynomial time even when the BranchyNet structure becomes complex.

REFERENCES

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[2] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.

[3] H. J. Jeong, "Lightweight offloading system for mobile edge computing," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops*, Kyoto, Japan, 2019, pp. 451–452.

[4] R. G. Pacheco and R. S. Couto, "Inference Time Optimization Using BranchyNet Partitioning," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, Rennes, France, 2020, pp. 1–6.

[5] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, Cancun, Mexico, 2016, pp. 2464–2469.

[6] X. Chen *et al.*, "DNNOff: Offloading DNN-Based Intelligent IoT Applications in Mobile Edge Computing," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 4, pp. 2820–2829, Aug. 2022.

[7] M. Xu *et al.*, "Unleashing the Power of Edge-Cloud Generative AI in Mobile Networks: A Survey of AIGC Services," *IEEE Communications Surveys & Tutorials*, vol. 26, no. 2, pp. 1127–1170, Secondquarter 2024.

[8] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wirel. Commun. Mob. Comput.*, vol. 13, pp. 1587–1611, 2013.

[9] B. Lin, Y. Huang, J. Zhang, J. Hu, X. Chen, and J. Li, "Cost-Driven Off-Loading for DNN-Based Applications Over Cloud, Edge, and End Devices," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 8, pp. 5456–5466, Aug. 2020.

[10] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun.*, 2018, pp. 31-36.

[11] M. A. Khan *et al.*, "A Survey on Mobile Edge Computing for Video Streaming: Opportunities and Challenges," *arXiv:2209.05761*, 2022.

[12] H. James Deva Koresh, "Quantization with Perception for Performance Improvement in HEVC for HDR Content," *Journal of Innovative Image Processing (JIIP)*, vol. 2, no. 01, 2020.

[13] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in neural information processing systems*, vol. 25, no. 2, 2012.

Edited by: Zhengyi Chai
Special issue on: Data-Driven Optimization Algorithms for Sustainable and Smart City
Received: Feb 22, 2024
Accepted: June 5, 2024